Temas Selectos de
Estructuras de Datos

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Febrero 2004

Índice general

Prefacio

Los *Temas Selectos de Estructuras de Datos* introducen en forma simple y sencilla a algunos temas relevantes de Estructuras de Datos. No tiene la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que por su relevancia se consideran esenciales para el conocimiento básico de esta área, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: Árboles de Búsqueda, Ordenamiento Secuencial, Almacenamiento por Hashing, Compresión de Texto, Búsqueda de Cadenas y Bases de Datos Relacionales. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe aprender en las asignaturas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Febrero 2004

Capítulo 1

Árboles de Búsqueda
Recorrido y Mantenimiento

Un *árbol* es una de las estructuras de datos más útiles que se han concebido en programación de sistemas. Consiste en un conjunto de nodos organizados que almacenan algún tipo de dato. Ciertamente, existen muchos algoritmos para la búsqueda y manipulación de los datos almacenados en árboles. Este capítulo describe cómo se construyen los árboles, un algoritmo de búsqueda de datos en el árbol, algunos esquemas de recorrer el árbol, y una técnica de mantener (añadiendo o borrando) los datos del árbol.

El uso de nodos y apuntadores es fundamental para la construcción de árboles de búsqueda. Un *nodo* no es mas que una colección de localidades de memoria asociadas en conjunto por un programa. Cada nodo tiene un nombre que, ya sea directa o indirectamente, se refiere a la dirección de una de sus localidades de memoria. Esta última normalmente tiene un contenido, que es el elemento a ser almacenado en el nodo. Además, incluye cero, uno o dos (y a veces mas) *apuntadores* o *ligas*. Estos apuntadores no son mas que los nombres de otros nodos en el árbol. Los programas que usan árboles de búsqueda tienen la opción de pasar de un nodo a otro siguiendo tales apuntadores.

Conceptualmente, un árbol de búsqueda se representa por cajas y flechas (figura 1.1). Cada caja representa un nodo, y cada flecha representa un apuntador. Cada nodo en el diagrama consiste de *campos* (espacios donde puede almacenarse un dato o apuntador), que en este caso, son un campo de datos y dos campos de apuntadores.

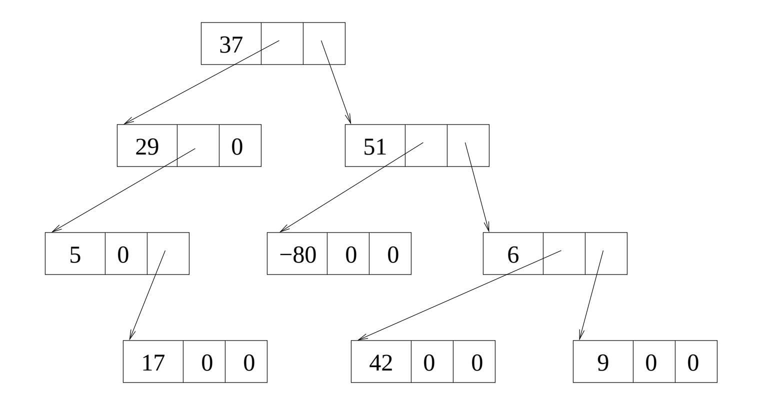


Figura 1.1: Un árbol de búsqueda con nueve nodos

Un árbol puede directamente implementarse en memoria principal mediante un programa a nivel ensamblador, o puede implementarse en forma de arreglo en un programa de alto nivel. La figura 1.2 ilustra ambas posibilidades para el árbol anterior.

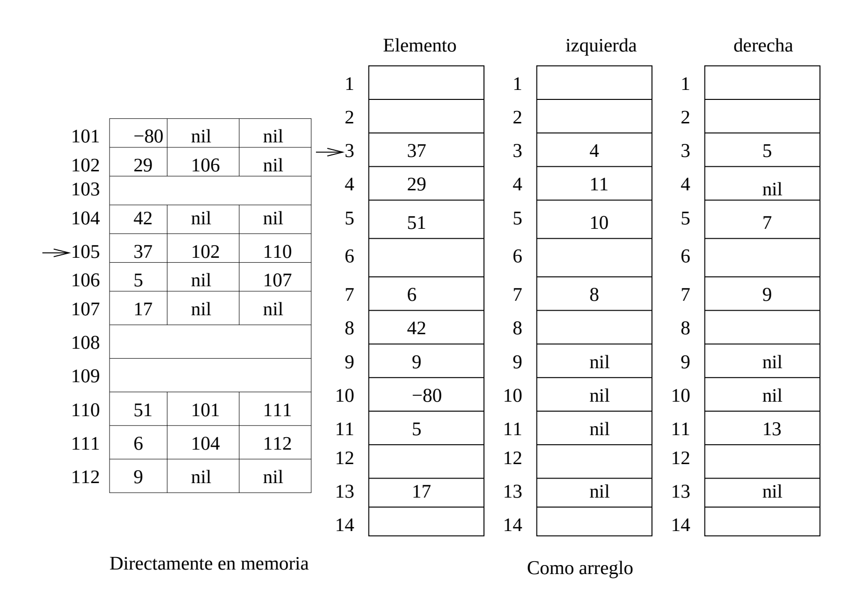


Figura 1.2: Dos formas de programar un árbol de búsqueda

No resulta difícil trazar en ambos diagramas los datos y apuntadores de un nodo a otro; meramente, se inicia en el nodo indicado con la flecha en ambos casos. En el primer caso, los tres campos se almacenan en una sola palabra en la dirección de memoria 105. El campo de datos contiene el valor 37, y los campos de apuntador indican respectivamente los nodos "hijos" izquierdo y derecho de ese nodo. En este caso, el nodo hijo izquierdo se encuentra en la dirección 102, y el nodo hijo derecho en la dirección 110. En la segunda representación, los campos de datos y apuntadores se encuentran almacenados en tres arreglos separados llamados Elemento, izquierda y derecha, respectivamente. En este caso, los campos son utilizados de la misma manera que en el caso anterior. Los campos de apuntador que no apuntan a nada contienen un valor "nil". Esto meramente significa que se usa un símbolo distinguible de valores (direcciones) válidos para los apuntadores. Los espacios en ambas representaciones se muestran para hacer evidente que los árboles son construidos en formas que el programador no siempre puede controlar.

Cuando los datos se almacenan en un árbol, frecuentemente es con el propósito de acceder a elementos particulares rápidamente. Por ejemplo, si los elementos que se almacenan tienen un orden en particular, como puede ser $<$ (menor que), entonces la búsqueda por elementos puede hacerse en forma especialmente rápida si los nodos se ordenan de acuerdo a la siguiente regla:

"*En cada nodo, todos los elementos almacenados en su subárbol izquierdo tienen un valor menor que el elemento almacenado en ese nodo; y todos los elementos almacenados en el subárbol derecho tienen un valor mayor que el elementos almacenado en ese nodo*" (figura 1.3)

Un árbol de datos que satisface tal condición se le conoce como *árbol de búsqueda binario*. La figura 1.4 muestra un ejemplo en el que puede verse qué tan rápido puede hacerse una búsqueda en árboles. En este caso, los elementos son nombres, y el ordenamiento simplemente es alfabético.

Para buscar en este árbol, se puede usar un algoritmo muy simple, que toma como entrada el "nombre", y su salida es *sí* o *no* dependiendo si tal nombre fue hallado en el árbol. Las notaciones $item(node)$, $left(node)$ y $right(node)$ se refieren, respectivamente, al elemento, al apuntador izquierdo y al apuntador derecho de un nodo dado. El algoritmo comienza en el nodo más alto del árbol:

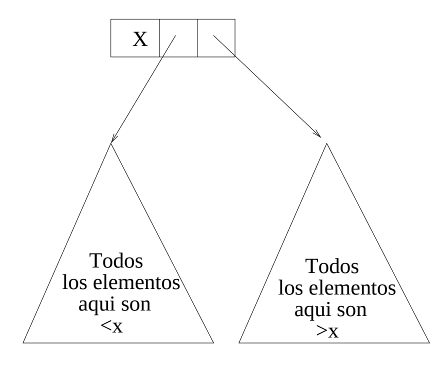


Figura 1.3: Un árbol de búsqueda puede organizarse mediante un orden

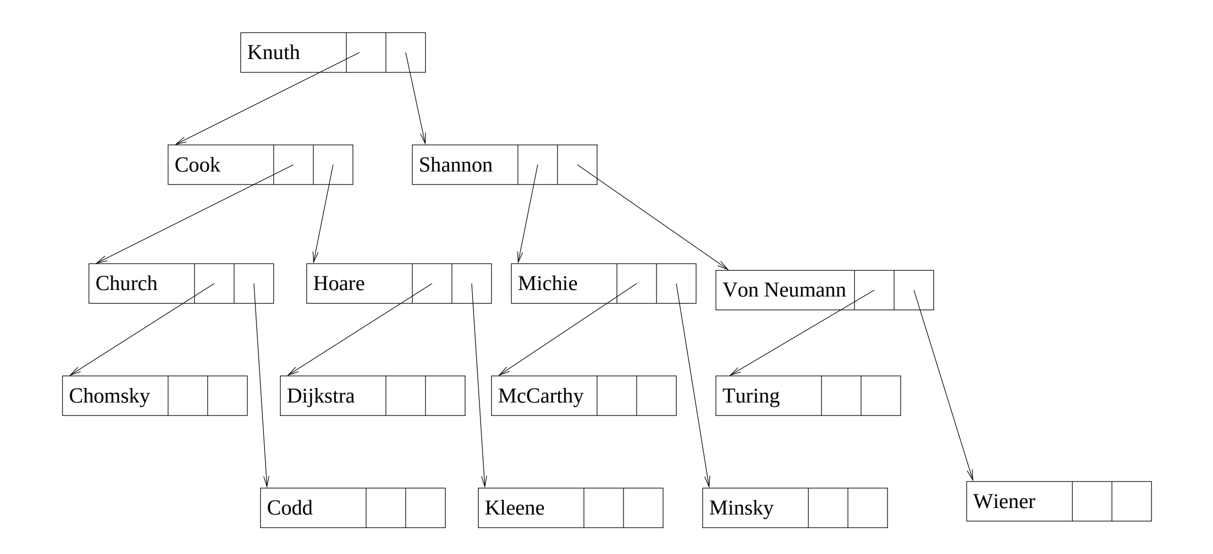


Figura 1.4: Utilizando nombres como datos almacenados en un árbol

procedure $SEARCH$

1. $found \leftarrow false$
2. $node \leftarrow top$
3. **repeat**
   a) **if** $item(node) = name$
      then $print\ si$; $node \leftarrow nil$; $found \leftarrow true$;
   b) **if** $item(node) > name$
      then $node \leftarrow right(node)$;
   c) **if** $item(node) < name$
      then $node \leftarrow left(node)$;
4. **until** $node = nil$;
5. **if not** $found$
   then $print\ no$;

El procedimiento $SEARCH$ desciende nodo a nodo: si el nombre que se busca es "menor que" el nombre almacenado en tal nodo, el algoritmo toma la rama izquierda; si el nombre que se busca es "mayor que" el nombre almacenado en el nodo actual, el algoritmo toma la rama derecha. Finalmente, si el nombre es encontrado en un nodo, el algoritmo imprime *sí*.

El ciclo termina tan pronto como $node = nil$. Esto sucede no sólo cuando el nombre buscado se encuentra, sino también cuando el algoritmo trata de descender a través de un apuntador nulo, es decir, ha llegado al fondo del árbol. Nótese el uso de la variable lógica $found$, que actúa como señal para la prueba, al final del algoritmo. Esta variable se inicializa con un valor de $false$, y sólo cambia si la búsqueda es exitosa.

Es notorio que el número de iteraciones en el ciclo principal de $SEARCH$ no es nunca mayor que el número de niveles en el árbol. Ya que la cantidad de trabajo hecho dentro del ciclo se limita a un número constante de pasos, es posbile escribir la complejidad en el tiempo de búsqueda limitado por $O(l)$, donde l es el número de niveles en el árbol. Sin embargo, en un árbol de búsqueda binario con n nodos, y todos los nodos excepto los más profundos teniendo dos hijos, el número de niveles l está dado por:

$$l = \lceil \log_2 n \rceil$$

Esto permite escribir la complejidad en el tiempo de $SEARCH$ como $O(\log n)$ en el caso general, al menos cuando el árbol se encuentra "lleno" en el sentido dado previamente.

Este tiempo de búsqueda es excesivamente rápido. Si en lugar de 15 de los más famosos personajes en computación se colocan mil millones de nombres de quienes trabajan en computación[1] el tiempo de búsqueda se incrementaría apenas de 4 pasos a sólo 30.

Ahora bien, puede darse el caso de que se requiera listar todos los datos almacenados en un árbol de búsqueda. Un algoritmo que produce tal lista puede recorrer el árbol, es decir, visitar cada nodo en alguno de varios ordenamientos, el más común de los cuales es el orden de "primero profundidad" (*depth-first order*) o preorden. Este modo de recorrido o visita se programa fácilmente utilizando la *recursión*. El algoritmo se mueve hacia arriba y abajo dentro del árbol siguiendo los apuntadores. La regla sencilla que define su progreso requiere, por ejemplo, que ningún nodo a la derecha de un nodo dado sea visitado hasta que todos los nodos a la izquierda hayan sido visitados.

procedure $DEPTH(x)$

1. $use x$
2. **if** $left(x) \neq nil$
   then $DEPTH(left(x))$
3. **if** $right(x) \neq nil$
   then $DEPTH(right(x))$

program $DEPTH(root)$

El paso "*use x*" significa aquí imprimir x o procesarlo de alguna manera. Si el procedimiento no ha alcanzado el fondo del árbol donde los apuntadores son nulos, entonces se llama a sí mismo primero descendiendo por la izquierda del nodo actual, y luego por la derecha. Obviamente, la llamada a $DEPTH$ con el nodo izquierdo como argumento podría resultar inmediatamente en otra llamada igual, y así hasta llegar al fondo del árbol. De esta forma, el algoritmo consiste en una sola llamada a $DEPTH$ con argumento en el nodo raíz (*root*) del árbol, lo que resulta en un barrido sistemático de izquierda a derecha através de todos los nodos del árbol.

[1] Lo cual sería suficiente para considerar a todos quienes trabajamos en computación

En muchas aplicaciones, los árboles de búsqueda binarios no se encuentran fijos de principio a fin, sino que pueden crecer o empequeñecerse como cualquier archivo de datos. Para insertar un nuevo elemento en un árbol de búsqueda (figura 1.5), es tan solo necesario modificar levemente el algoritmo de búsqueda: tan pronto como la búsqueda por el elemento falla, obténgase una dirección nueva (no usada), y reemplácese el apuntador a nulo encontrado con tal dirección. En seguida se coloca el nuevo elemento en la dirección, y se crean dos apuntadores nulos, para formar entre los tres los campos de un nuevo nodo.

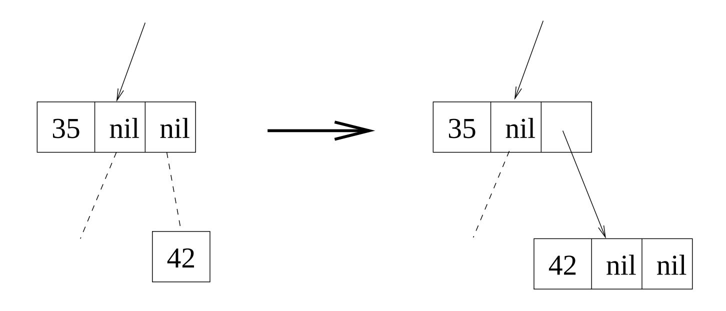


Figura 1.5: Añadiendo un nodo

Borrar un nodo resulta algo más complicado. De nuevo, el algoritmo de búsqueda puede ser utilizado para localizar un elemento, pero la dirección del nodo que apunta hacia él debe preservarse por el algoritmo de borrado. En la segunda parte de la figura 1.6 un algoritmo de borrado se muestra en operación.

Inspeccionando la figura 1.6, es notorio que el nodo conteniendo el valor 42 se remueve del árbol conjuntamente con sus dos apuntadores. Consecuentemente, otros dos apuntadores (marcados con un asterisco) deben ser re-arreglados. Esencialmente, todo el subárbol derecho dependiente del nodo conteniendo el valor 42 se mueve hacia arriba un nivel, mientras que todo el subárbol izquierdo se ha deslizado hacia abajo dos niveles hasta el primer nodo disponible en la posición más a la izquierda del primer subárbol. La palabra “disponible” aquí se refiere a que el apuntador izquierdo del nodo es nulo y es posible colocar el segundo subárbol ahí.

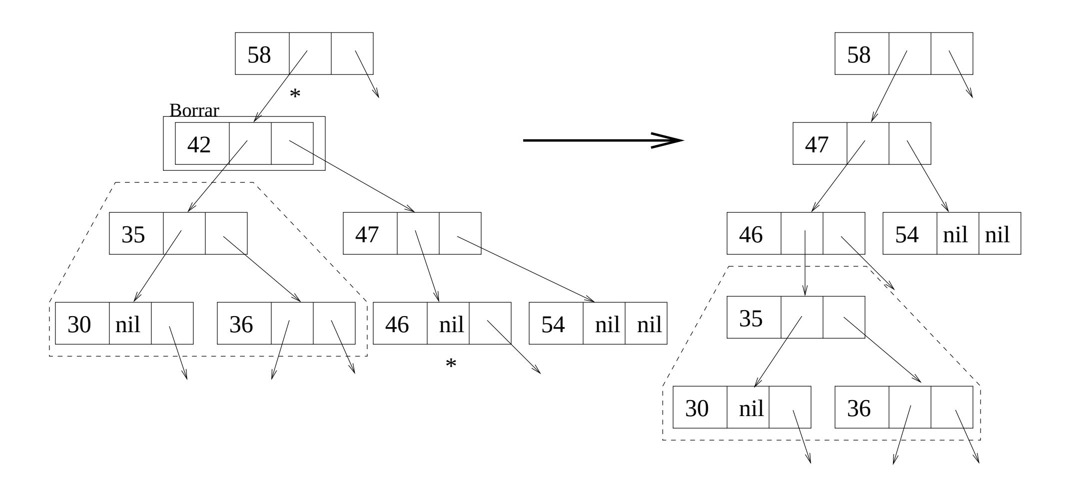


Figura 1.6: Borrando un nodo

Capítulo 2

Ordenamiento Secuencial
Un Límite Inferior de Velocidad

Ordenar es una tarea computacional con enormes implicaciones en el comercio y la industria. Muchas de las computadoras de uso comercial e industrial pasan una fracción significativa de su tiempo ordenando listas. De este modo, resulta muy práctico preguntarse: *¿Dada una lista de n elementos, cuál es la mínima cantidad de tiempo que puede llevarle a una computadora ordenarla?*

Un algoritmo conocido como mezcla-ordenamiento (*merge-sort*) puede lograr esta tarea en un tiempo $O(n \log n)$. Partiendo de esto, y considerando una n lo suficientemente grande, es posible proponer una constante c positiva tal que el número de pasos que requiere este algoritmo para ordenar n elementos nunca es mayor que $cn \log n$. Esta cantidad tiene también el orden de magnitud correcto para ser un *límite inferior* de el número de pasos requerido, al menos cuando la pregunta anterior se reformula adecuadamente.

Si la palabra "ordenarla" se restringe a significar que la computadora sólo compara dos elementos de la lista de n elementos en un momento dado, entonces este límite inferior no es difícil de demostrar. Sea la lista a ser ordenada representada como:

$$L = x_1, x_2, ..., x_n$$

Considérese que la computadora comienza comparando una x_i con una x_j, y hace una de dos cosas como resultado de la comparación (figura 2.1).

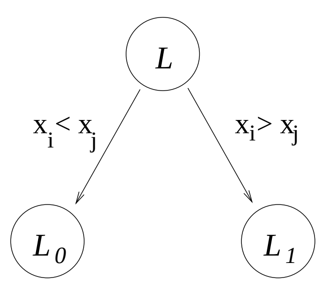


Figura 2.1: Decisión inicial de un algoritmo de ordenamiento

Lo que se considera hace la computadora es alterar la lista L para convertirla en L_0 o L_1, dependiendo de la comparación. Por ejemplo, a partir de L la computadora puede intercambiar x_i y x_j para crear L_0, o dejar x_i y x_j en su posición original para crear L_1. De modo similar, la computadora puede continuar trabajando sobre L_0 o L_1, aplicando precisamente la misma observación. Continuando con este proceso, resulta un árbol cuyos nodos más bajos corresponden todos a una versión ordenada de la lista inicial.

Por ejemplo si L tiene sólo tres elementos, la figura 2.2 muestra como el "árbol de decisión" podría ser.

Para entender la significancia de los nodos terminales, es útil imaginarse qué hace el algoritmo de ordenamiento con todas las $n!$ posibles versiones de listas de un conjunto de n elementos. Dadas dos de tales listas, supóngase que el algoritmo toma la misma rama para ambas listas en cada nivel del árbol, de tal modo que se llega al mismo nodo terminal en cada caso. Cada vez que el algoritmo de ordenamiento toma una decisión como resultado de una comparación, se realiza una permutación π de la versión actual de la lista de entrada L. Más aun, se realiza esta permutación en cualquier lista que se tenga actualmente. Por lo tanto, si dos versiones distintas de la lista de un mismo conjunto terminan en el mismo nodo terminal, entonces ambas listas han permutado de forma idéntica. Pero esto significa que a lo mas, una de esta listas ha sido ordenada correctamente. La única conclusión que surge de todo esto es que si el algoritmo de ordenamiento se realiza correctamente, el árbol de decisión correspondiente debe tener al menos $n!$ nodos terminales.

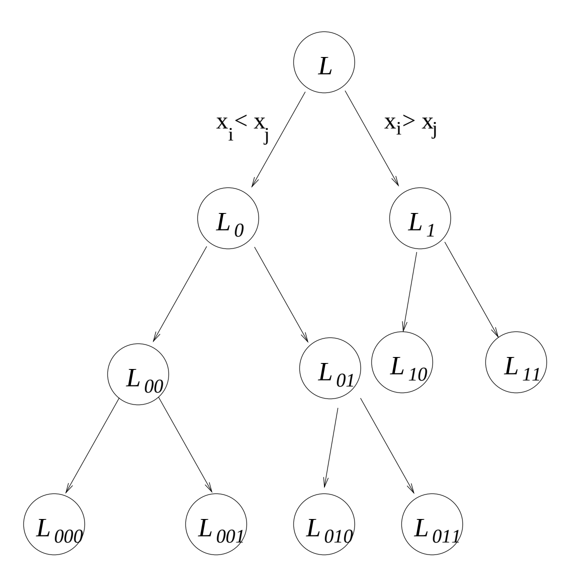


Figura 2.2: Un árbol de decisión para una lista de tres elementos

Ahora bien, un árbol binario con m nodos terminales debe tener una profundidad de al menos $\lceil \log m \rceil$, y la profundidad (D) del árbol de decisión debe satisfacer que:

$$
\begin{aligned}
D &= \lceil \log m \rceil \\
&= \log n(n-1)(n-2)...(n/2) \\
&> \log(\frac{n}{2})^{n/2} \\
&= \frac{n}{2} \log \frac{n}{2} \\
&= O(n \log n)
\end{aligned}
$$

Esto es tan solo otra forma de decir que cualquier comparación para ordenamiento que tome una lista de n elementos como entrada debe realizar al menos $O(n \log n)$ comparaciones en el peor de los casos. Cualquier algoritmo que se realice substancialmente mejor que esto, en total, debe hacer su decisión para ordenamiento en base a otro criterio, o debe ser capaz de hacer las comparaciones en paralelo.

Capítulo 3

Almacenamiento por Hashing

La Clave está en la Dirección

Hay principalmente tres técnicas de almacenar y recobrar registros en archivos grandes. La más simple involucra una búsqueda secuencial a través de los n registros, y requiere $O(n)$ pasos para recobrar un registro particular. Si los n registros están especialmente ordenados o almacenados en un árbol, entonces un registro puede recobrarse en $O(\log n)$ pasos (vésae el capítulo 1). Finalmente, si cierta información de cada registro se utiliza para generar una dirección de memoria, entonces un registro puede recobrarse promedio en $O(1)$ pasos. Esta última técnica, que es el tema de este capítulo, se le conoce como *hashing*.

Considérese un registro "típico" en un archivo, consistente de una *llave* y un *dato*, como se muestra en el siguiente ejemplo:

3782-A:670-15 DURALL RADIAL, 87.50, STOCK 24

Aquí, la llave es 3782-A, y el dato consiste de un tamaño y marca de llanta seguido por su precio por unidad y número de elementos en almacenamiento. Respecto a almacenar, buscar y recobrar registros de archivos, es suficiente especificar cómo manipular la llave; el dato se "añade" dada la programación apropiada.

Transformar una llave (o *hashing*), en un sentido, es cortarla y usar sólo parte de ella. La parte usada directamente genera una dirección de memoria,

Llave	ASCII	Direccion
TO	124117	17
IT	111124	24
IS	111123	23
AS	101123	23
AT	101124	24
IF	111106	06
OF	117106	06
AM	101115	15
BE	102105	05
DO	104117	17
AN	101116	16
GO	107117	17
SO	123117	17

05	BE
06	IF, OF
07	
10	
11	
⋮	
14	
15	AM
16	AN
17	TO, DO, GO
⋮	
23	AS, IS
24	IT, AT
25	

Figura 3.1: Transformación de llaves en dígitos

como se muestra en la figura 3.1, la cual usa palabras de dos letras en inglés como llaves.

Los dos últimos dígitos del código ASCII para cada llave k se utilizan como dirección $h(k)$, en la cual cada palabra puede ser almacenada. De acuerdo con esto, en la parte derecha de la figura 3.1 se muestra una porción de memoria en forma esquemática. Además de cada localidad de memoria, se muestra la llave que ha sido transformada a tal dirección. Algunas localidades de memoria no tienen llaves, y otras tienen más de una.

Esto último se conoce como *colisión*, y para hacer que el *hashing* trabaje correctamente, las colisiones deben resolverse. Uno podría pensar que las colisiones son relativamente raras. Ciertamente, el gran número de colisiones en el ejemplo previo se debe al hecho de que varias de las palabras de dos letras usadas como llaves terminan en la misma letra. De hecho, como lo apunta Knuth, las colisiones son casi la regla, aun con llaves distribuidas aleatoriamente, mucho antes de que el espacio de memoria reservado para el almacenamiento sea totalmente utilizado. Knuth ilustra este punto con la famosa *paradoja del cumpleaños*: ¿Cuál es la probabilidad de que al menos dos personas en una habitación con 23 personas tengan el mismo cumpleaños?

La respuesta es más que 0.5. Si se considera a los 365 posibles cumpleaños como localidades de memoria, a los nombres de las 23 personas como datos, y sus cumpleaños como llaves, entonces este simple ejemplo muestra qué tan posibles son las colisiones: la probabilidad de que al menos suceda una colisión es mayor que 0.5 aun antes de que el 10 por ciento de las localidades de memoria sean utilizadas.

Existen principalmente dos métodos para resolver las colisiones, llamados encadenamiento (*chaining*) y área de desbordamiento (*open addresing*). Sin embargo, antes de describirlos, se observa qué puede lograrse mediante mantener las colisones a un mínimo.

El medio de almacenamiento se considera como un conjunto de direcciones enteras de 0 a M; éstas pueden ser índices de un arreglo o direcciones verdaderas de memoria, dependiendo del nivel del lenguaje usado. Una llave k es generalmente una cadena alfanumérica, pero siempre podemos convertir k en su equivalente entero ASCII. Suponiendo que k es ya un entero, ¿cómo generar una dirección de memoria en un rango de 0 a M a fin de minimizar la probabilidad de colisiones? Una forma es utilizar la operación módulo de M:

$$h(k) = k \, mod \, M$$

La única forma en que se puede controlar la función h es seleccionando M; no es una cuestión de qué tan grande o pequeña hacer M. Es solo necesario que M sea el *tipo* correcto de número. Se ha observado que si M es un número primo, entonces h genera direcciones de memoria bien distribuidas, siempre y cuando M no tenga la forma $r^k + a$, donde r es la raíz del conjunto de caracteres ($r = 128$ para los caracteres ASCII), y a es un entero pequeño. Si se toma este consejo para el ejemplo anterior, entonces es notorio que $M = 23$ está ciertamente muy lejos de la forma 128^k.

Con este mismo valor de M se obtienen muy diferentes resultados para las mismas 13 palabras de dos letras. Las direcciones de memoria en la figura 3.2 se expresan en notación decimal. Con la primera función *hash* se sufrían cuatro colisiones (contando TO, DO y GO como dos colisiones), pero con la nueva función *hash* este número baja a tan solo dos.

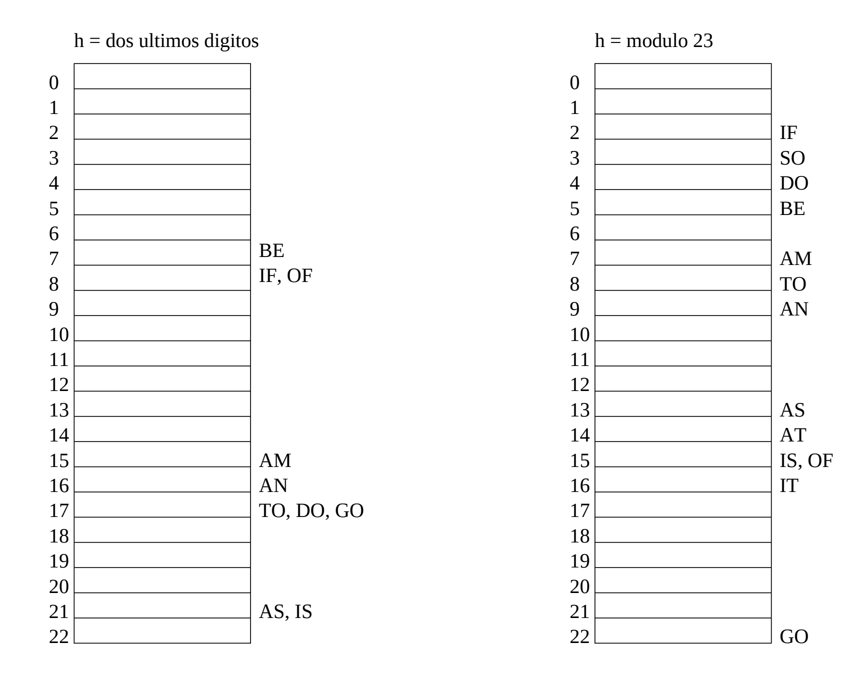


Figura 3.2: Utilizando la función modulo con primos genera una mejora

Además del método de la división, hay otro método útil el cual usa la multiplicación. Se basa en la observación general de que si uno toma un número irracional x, y forma n de sus múltiplos:

$$x, 2x, 3x, ..., nx$$

y tomando sus partes fraccionales hasta n:

$$x_1, x_2, x_3, ..., x_n$$

los número resultantes son todos diferentes y dividen al intervalo unitario en $n + 1$ subintervalos. Cada uno de estos subintervalos tiene al menos uno de tres posibles longitudes, y si se añade un nuevo punto x_{n+1}, éste cae en un subintervalo del tipo más grande. Sucede que si x se escoge a ser la "media áurea" (con el valor aproximadamente de 0.61803399), entonces las tres longitudes sufren la menor variación, y todas están más cercanas que las producidas por cualquier otro número irracional. Estas observaciones sugieren una función *hash* h producida de acuerdo al siguiente algoritmo.

Sea g la razón áurea, tan cercana como el tamaño de una palabra pueda aproximarla.

1. $h \leftarrow k \cdot g$
2. $h \leftarrow parte\, fraccional\, de\, h$
3. $h \leftarrow h \cdot M$
4. $h \leftarrow parte\, entera\, de\, h$

Este cálculo forma el producto $k \cdot g$, toma su parte fraccionaria, lo escala a M y toma el entero más cercano (menor que) el resultado.

Aplicando este método al ejemplo de las llaves para palabras de dos letras, solo se obtiene una colisión, lo cual es el mejor desempeño que se pudiera esperar bajo las circunstancias actuales.

Ahora bien, habiendo discutido algunos tipos de funciones *hash*, se retoman las técnicas para manejo de colisiones. La primera involucra la construcción de una cadena de apuntadores desde cada dirección en la cual ocurra una colisión, y la segunda se refiere a cambiarse a una nueva dirección de memoria dentro de la tabla *hash* en la porción de memoria dedicada al almacenamiento de llaves.

El encadenamiento se ilustra en la figura 3.3 para la función *hash* del ejemplo.

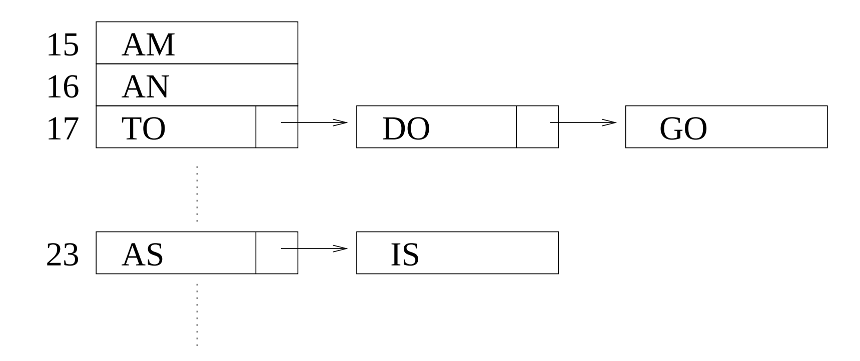


Figura 3.3: Encadenamiento (*chaining*)

Las llaves TO, DO y GO se asocian todas con la dirección 17, en ese orden. Las llaves AS e IS se asocian con la dirección 23. Si una porción

de cada localidad de memoria en la tabla se reserva para un apuntador, entonces cuando una llave como DO se asocia con la misma dirección que TO, una nueva localidad se añade de un espacio de memoria auxiliar. La llave DO se coloca en esta localidad, y su dirección se inserta en el espacio para apuntador asociado con la primera palabra TO. Similarmente, cuando después llega GO a la misma dirección, se le añade a la cadena tras DO mediante un segundo apuntador.

La otra técnica para resolver colisiones, llamada área de desbordamiento, consiste en insertar una llave k, y descubrir que $h(k)$ se encuentra ya ocupada por otra llave; entonces, se "prueba" la tabla *hash*, examinando direcciones a una distancia fija $p(k)$ adelante de la dirección que se considera actualmente. Si $h(k)$ está ocupada, se examina $h(k)-p(k)$, y si ésta se encuentra ocupada, se examina $h(k) - 2p(k)$, y así sucesivamente.

Si $p(k) = 1$, la técnica lineal resulta en un largos "racimos" (*clusters*) conforme se llena la tabla. Al hacer $p(k) = c$, donde c es un entero "relativamente primo" (i.e. con pocos divisores) respecto a M, hay una tendencia menor a formarse racimos.

Quizá la mejor técnica de todas es combinar esta prueba secuencial con una tabla *hash* ordenada: si las llaves k se han insertado en la tabla en orden decreciente de k, entonces el siguiente algoritmo inserta una nueva llave en la tabla, siempre y cuando haya espacio para ella.

1. $i \leftarrow h(k)$
2. **if** $contenido(i) = 0$ **then** $contenido(i) \leftarrow k$; **exit**
3. **if** $contenido(i) < k$ **then** *intercambia valores de* $contenido(i)$ *y* k
4. $i \leftarrow i - p \ mod \ M$
5. **goto** 2

Es interesante notar que si esta política de inserción se sigue desde el inicio de la construcción de la tabla, produciría la misma tabla que se ha supuesto. Es decir, este algoritmo tanto introduce como mantiene la misma tabla que se obtuvo al insertar primero la llave más grande, después la siguiente más grande, y así sucesivamente.

Claramente, se hace más con una tabla *hash* que meramente hacer inserciones. Más frecuentemente, se busca en la tabla para ver si una llave se encuentra ahí. Ocasionalmente, también se borran elementos de la tabla.

En la mayoría de los casos, buscar en una tabla *hash* es muy similar a insertar una nueva llave: se calcula el valor *hash* $h(k)$, y se verifica la dirección para ver si k se encuentra en la tabla. Dependiendo si se trata de encadenamiento o de área de desbordamiento, se sigue una cadena de apuntadores o se prueba una secuencia dentro de la tabla.

Si m direcciones de las M direcciones en una tabla contienen llaves , y las cadenas de tal tabla contienen además n llaves, entonces el número de pasos requeridos para buscar en una tabla con encadenamiento es n/m. Si la función *hash* lleva a una distribución relativamente uniforme de llaves, entonces m es mayor que n, y el tiempo de búsqueda es esencialmente constante (y pequeño).

La razón m/M se conoce como "factor de carga" (*load factor*) de una tabla *hash*. Bajo la técnica de área de desbordamiento, una secuencia de pruebas requiere de

$$O\left(\frac{M}{M-m}\right)$$

pasos para localizar una llave dentro de la tabla. Aun cuando este resultado puede ser derivado teóricamente, hasta ahora solo hay evidencia empírica que el orden de una tabla *hash* requiere de

$$O\left(\frac{M}{m}\log\frac{M}{M-m}\right)$$

pasos para una búsqueda exitosa. Este límite de complejidad promedio es, sin embargo, una conjetura. Nótese que la complejidad de una prueba lineal crece mucho más rápidamente que el orden supuesto de la complejidad conforme m se aproxima a M.

Resulta útil comparar varias técnicas de almacenamiento y recuperación para decidir cuál es la mejor para una aplicación dada. Por ejemplo, es posible comparar las tablas *hash* con los métodos de árboles binarios (véase el capítulo 1), y concluir que aun cuando el tiempo de acceso es mucho menor en las tablas *hash*, esto se debe a que generalmente se utiliza una tabla de tamaño fijo M. Esta inflexibilidad puede ser una desventaja si el archivo que se mantiene no tiene un límite específico de crecimiento. Sin embargo, las tablas *hash* son más fáciles de programar que algunos esquemas basados en árboles.

Capítulo 4

Compresión de Texto
Codificación Huffman

Las dos aplicaciones más importantes de las técnicas de codificación son la protección y compresión. Cuando un mensaje es codificado como una cadena de ceros y unos, existen técnicas que, mediante la inserción de algunos bits extra en la cadena a trasmitir, permiten al receptor del mensaje descubrir cuáles ceros o unos (si los hay) son erróneos. Además, hay métodos de acortamiento de la cadena, de tal modo que no se requieren transmitir muchos bits.

Las dos grandes áreas de aplicación de la teoría de codificación son *tiempo* y *espacio*. Las aplicaciones de tiempo son aquellas de comunicación tradicional en que un mensaje se transmite electrónicamente a través del tiempo, y la mayor preocupación es proteger el mensaje de errores. Las aplicaciones de espacio involucran la protección o compresión de datos durante el almacenamiento en algún medio electrónico, como la memoria de una computadora o los discos magnéticos.

Si una gran cantidad de texto debe almacenarse, o si el espacio de almacenamiento es importante, conviene comprimir el texto de alguna forma antes de almacenarlo. La codificación Huffman hace esto mediante explotar la redundancia en el texto fuente.

En esencia, la idea es muy simple. Supóngase que se tiene una cadena de texto que usa un alfabeto con símbolos s_1, s_2, ..., s_n, y que la probabilidad de que el i-ésimo símbolo que aparece en un punto aleatoriamente seleccionado de la cadena es p_i. Cada símbolo s_i se substituye por una cadena binaria de

longitud l_i, de tal modo que la longitud promedio que representa un símbolo está dado por

$$L = \sum_{i=1}^{n} p_i l_i$$

Suponiendo que $p_1 > p_2 > ... > p_n$, se nota que L se minimiza sólo si $l_1 < l_2 ... < l_n$. Esta observación forma la base para una técnica que selecciona las cadenas binarias reales que codifican los varios símbolos. Supóngase, por ejemplo, que los símbolos A, B, C, D, E, F y G aparecen con las probabilidades 0.25, 0.21, 0.18, 0.14, 0.09, 0.07 y 0.06 respectivamente. Nótese que la suma de todas las probabilidades suma 1.0.

Un *árbol de Huffman* para estos símbolos y sus probabilidades es una representación visual conveniente de la selección, en la cual las cadenas binarias representan los símbolos (figura 4.1). Cada nodo del árbol está etiquetado con la suma de las probabilidades asignadas a los nodos que tiene debajo.

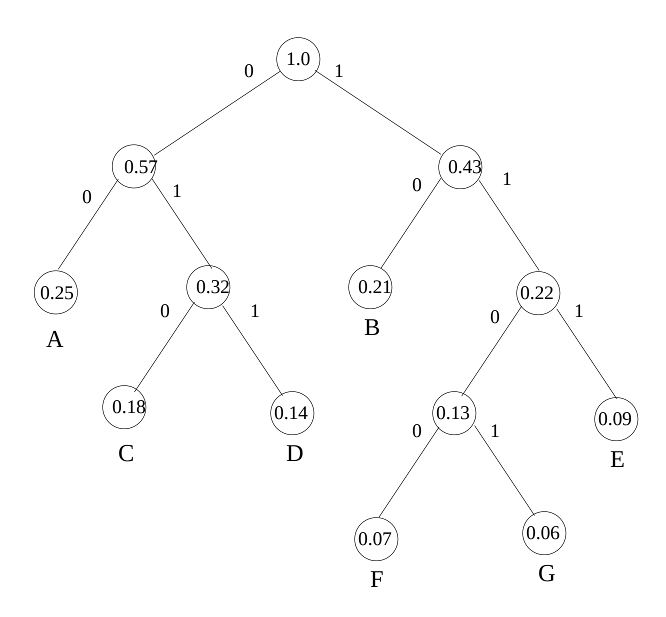


Figura 4.1: Un árbol de Huffman

Antes de describir cómo se construye este árbol, nótese que cada símbolo en el alfabeto fuente se encuentra en un nodo terminal del árbol, y que la cadena binaria que lo codifica está en la secuencia de ceros y unos que se

encuentran al recorrer el árbol desde la raíz a un nodo en particular. La codificación Huffman para el sistema que se ha descrito es por lo tanto:

A	00
B	10
C	010
D	011
E	111
F	1100
G	1101

La longitud promedio del código L es 2.67 bits, y éste es un mínimo.

El algoritmo que asigna estos códigos construye un árbol de Huffman explícito de abajo hacia arriba. Un arreglo llamado "*subárboles*" contiene los índices de las raíces de todos los subárboles de los que se construye el árbol de Huffman. Inicialmente, *subárboles* no contiene más que subárboles de un solo nodo, uno por cada uno de los n símbolos s_i del alfabeto que se codifica. Hay, por lo tanto, un arreglo correspondiente de probabilidades "*prob*", que por cada nodo raíz en cada subárbol en construcción, contiene la probabilidad asociada con ese nodo. Dos arreglos de apuntadores "*ligaizquierda*" y "*ligaderecha*" contienen la estructura del árbol de Huffman, y cuando el algoritmo se termina, debe haber $2n-1$ nodos, de los cuales n son nodos y el resto apuntadores.

A la i-ésima iteración del algoritmo, los primeros $n-i+1$ subárboles se reordenan, de tal modo que sus probabilidades asociadas formen una secuencia decreciente. Los dígitos binarios apropiados se añaden a los nodos terminales de los últimos dos subárboles en secuencia reordenada, y éstos se mezclan para formar un nuevo subárbol que se coloca en el arreglo *subárboles*.

for $i \leftarrow 1$ **to** $n-1$

1. reordena *subárboles* desde 1 a $n-i+1$

2. añade 0 a la terminal del subárbol $n-i$

3. añade 1 a la terminal del subárbol $n-i+1$

4. $prob(n+i) \leftarrow prob(sub\acute{a}rboles(n-i)) + prob(sub\acute{a}rboles(n-i+1))$

5. $ligaderecha(n+i) \leftarrow n-i+1$

6. $ligaizquierda(n+i) \leftarrow n-i$

7. *subárboles*$(n-i) \leftarrow n+i$

El proceso que se indica por el paso "añade 0 a la terminal del subárbol $n-i$" simplemente quiere decir recorrer el $(n-i)$-ésimo subárbol y añadir un 0 a cada palabra almacenada como sus nodos terminales. No se hace referencia explícita de estas palabras en el algoritmo, pero son fácilmente manejadas cuando el algoritmo se convierte en un programa. El paso " añade 1 a la terminal del subárbol $n-i+1$" tiene un significado similar.

Después de que los apuntadores se modifican para asociarse con el nuevo nodo $n+i$ (siendo los nodos anteriores $1, 2, ..., n, n+1, ..., n+i-1$), el algoritmo finalmente reemplaza las dos últimas entradas en el arreglo *subárboles* por la raíz del nuevo subárbol construido. Cuando i llega a ser $n-1$, hay un solo subárbol en todo el arreglo, y éste es el árbol de Huffman.

Los primeros pasos de la operación del algoritmo pueden ilustrarse por el ejemplo ya presentado. Inicialmente hay $n-i+1 = 7$ subárboles que consisten de un nodo cada uno. Esto se rearreglan en orden decreciente respecto a su probabilidad (figura 4.2).

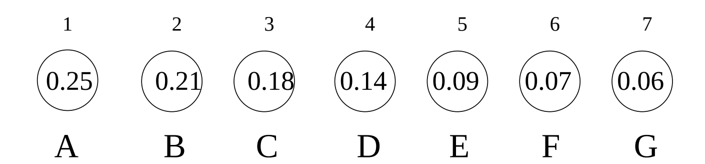


Figura 4.2: Siete subárboles de un solo nodo

Después de que un 0 y un 1 se añaden a las terminales de los subárboles 6 y 7, se crea un nuevo subárbol con índice $n+i = 8$. Éste se liga a la derecha con el subárbol 7 y a la izquierda con el subárbol 6. A la siguiente iteración del algoritmo, los subárboles se rearreglan en orden decreciente de probabilidad (figura 4.3).

A la siguiente iteración, un nuevo subárbol se forma de los dos anteriores, y el número de subárboles se reduce a 5 (figura 4.4).

Después de $n = 7$ iteraciones, sólo queda un subárbol, el cual es el árbol de Huffman. A cada paso, se asignan 0 o 1 como dígitos del código a la liga izquierda y derecha respectivamente, en forma sucesiva a los subárboles con probabilidades menores. Esto asegura que las palabras más largas de la codificación tiendan a ser menos frecuentemente utilizadas.

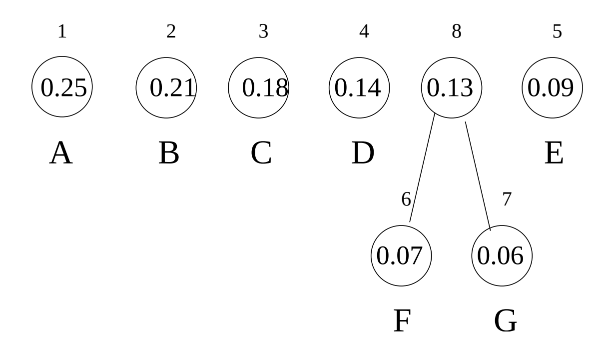


Figura 4.3: Subárboles después de la primera iteración

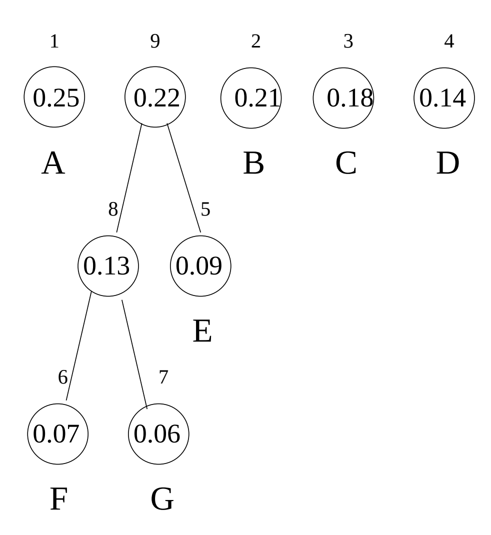


Figura 4.4: Subárboles después de la segunda iteración

Para almacenar una pieza dada de texto utilizando la codificación Huffman, se crea una tabla en la cual se cuenta el número de veces n_i que el i-ésimo símbolo ocurre. La probabilidad p_i de la pieza de texto es simplemente n_i/l, donde l es la longitud de la pieza de texto. Aplicar el algoritmo resulta en la asignación de un código a cada símbolo, así como en la construcción del árbol de Huffman. El texto se recorre, y en una sola pasada, se convierte en una larga cadena binaria de códigos concatenados. En seguida, esta cadena se divide en bloques de longitud m, que se refiere al tamaño de palabra de la computadora particular para la que el algoritmo se implementa. Cada bloque, entonces, se convierte en un solo entero, y se almacena en la memoria de esta forma. El árbol de Huffman, por su parte, se almacena en otro sitio de la memoria.

Para recuperar el texto que ha sido así almacenado, se invoca el procedimiento inverso, y el árbol de Huffman se utiliza en el último paso del procesamiento para los símbolos de la cadena binaria. Cada símbolo corresponde a una sola sub-cadena, determinada por una sola búsqueda desde la raíz hasta el nodo terminal pertinente del árbol.

Para evitar construir un árbol cada vez que algún texto se almacena, es posible construir un árbol más general de una vez por todas, que refleje las probabilidades de cada símbolo en forma más genérica. Por ejemplo, la oración promedio en Inglés (ignorando la puntuación) contiene los símbolos de A a Z y un espacio en blanco (β) en una localidad aleatoria con las siguientes probabilidades

A	0.065	B	0.013	C	0.022
D	0.032	E	0.104	F	0.021
G	0.015	H	0.047	I	0.058
J	0.001	K	0.005	L	0.032
M	0.032	N	0.058	O	0.064
P	0.015	Q	0.001	R	0.049
S	0.056	T	0.081	U	0.023
V	0.008	W	0.018	X	0.001
Y	0.017	Z	0.001	β	0.172

Capítulo 5

Búsqueda de Cadenas
El Algoritmo Boyer-Moore

Existen algunas sutilezas en lo que parece ser el asunto sencillo de buscar un patrón particular dentro de una cadena de caracteres, especialmente si desea hacerse rápidamente. Por lo tanto, la pregunta *¿Qué tan rápido se puede buscar un patrón dado dentro de una cadena de n caracteres?* tiene una cierta dificultad práctica.

Por ejemplo, como todo minero sabe, hay muchas formas de buscar ORO en LAS MINAS. Una cantidad razonable de métodos dictan que se busque la cadena en un orden definido, por ejemplo, de izquierda a derecha. Se intenta hacer coincidir el patrón con la parte más a la izquierda de la cadena, y si la coincidencia falla, se continúa recorriendo la cadena hacia la derecha hasta que se encuentra una coincidencia o se acaba la cadena. Realmente, parece razonable comparar los primeros caracteres en el patrón y la cadena en tal orden:

```
LAS MINAS
ORO
```

Ya que O y L no coinciden, no es necesario en continuar las comparaciones, por lo que se recorre el patrón en una unidad a la derecha, y se intenta de nuevo:

```
LAS MINAS
 ORO
```

De nuevo, los caracteres comparados, A y O, no coinciden. Recorriendo de esta forma, eventualmente la cadena se acaba, y se concluye que no hay ORO en LAS MINAS. Varios editores de texto hacen búsquedas de cadenas de esta forma.

Un algoritmo descubierto por R.S. Boyer y J.S. Moore en 1977 mejoran este método mediante verificar la coincidencia de los caracteres de derecha a izquierda en lugar de izquierda a derecha. Si la sección de la cadena no coincide con el patrón, se puede recorrer el patrón por toda su longitud. De este modo, se puede ir de:

```
LAS MINAS
ORO
```

inmediatamente a:

```
LAS MINAS
   ORO
```

A primera vista, parece ser un dramático incremento en la eficiencia, especialmente si en el siguiente paso el patrón puede saltar otros tres caracteres por la misma razón. Desafortunadamente, este incremento en la eficiencia es meramente ilusorio, ya que para verificar si A no está en ORO requiere tantas comparaciones como el recorrer ORO hacia adelante un espacio cada vez después de una comparación de la primera O con cada uno de los tres caracteres en la cadena.

Sin embargo, si se construye una tabla (llamada *tabla 1*) la cual por cada letra del alfabeto contiene su posición más a la derecha en el patrón ORO, entonces sólo es necesario buscar A en la tabla (lo que se puede hacer muy rápidamente) y notar que su entrada correspondiente es nula (no existe A en ORO, lo que se simboliza por $\emptyset$). Uno puede entonces recorrer el patrón a la derecha por tantos caracteres como los haya en él.

Además de la técnica de recorrer basado en *tabla 1*, el algoritmo Boyer-Moore usa otra buena idea: supóngase que en el proceso de hacer coincidir los caracteres del patrón con los caracteres de la cadena, los primeros m caracteres (de derecha a izquierda) se encuentra que coinciden:

```
                    ↓↓↓
TRESTRISTESTIGRESTRAGABANTRIGO
               TRASTRA
                    ↑↑↑
```

En tal caso, la porción que concide hasta ahora al final del patrón podría bien ocurrir en cualquier otro lado del patrón. Sería entonces inteligente recorrer el patrón a la derecha sólo la distancia entre las dos porciones:

```
                    ↓↓↓
TRESTRISTESTIGRESTRAGABANTRIGO
                    TRASTRA
                    ↑↑↑
```

En el ejemplo anterior, el algoritmo Boyer-Moore podría descubrir rápidamente que B no está en TRASTRA, y como resultado, recorrere el patrón por otros siete caracteres.

El algoritmo Boyer-Moore es corto y directo. Además de *tabla 1*, utiliza otra tabla, *tabla 2*, que por cada porción terminal del patrón indica su recurrencia más a la derecha y no terminal. Esta tabla puede accederse también rápidamente.

STRING

1. $i \leftarrow largo$
2. **while** haya cadena
   a) $j \leftarrow ancho$
   b) **if** $j = 0$ **then** *imprime* 'Coincidencia en' $i + 1$
   c) **if** $string(i) = pattern(j)$ **then** $j \leftarrow j - 1$, $i \leftarrow i - 1$, **goto** 2.b

d) $i \leftarrow i + max(table1(string(i)), table2(j))$

donde *ancho* es la longitud del patrón, *largo* es la longitud de la cadena, i es la posición actual en la cadena de caracteres, j es la posición actual en el patrón, *string(i)* es el i-ésimo caracter en la cadena, y *pattern(j)* es el j-ésimo caracter en el patrón.

Como ejemplo del algoritmo anterior en operación, supóngase que el patrón TRASTRA se ha colocado en seguida de ...GRESTRA... en la cadena. Comenzando en el paso 2.c, se nota que $i = 20$ y $j = 7$. Ya que *string(26)* = *pattern(7)*, ambos i y j se decrementan. Retornando al paso 2.b con $i = 19$ y $j = 6$ se encuentra otra coincidencia, volviendo $i = 18$ y $j = 5$. Finalmente, una nueva concidencia sucede, lo que deja $i = 17$ y $j = 4$. Sin embargo, los caracteres en las siguientes posiciones no coinciden, y la ejecución procede con *string(17)* = *E*. Por otro lado, la entrada en *tabla 1* aparece como sigue:

C	2
D	2
E	**5**
F	6
G	6

Por tanto, *tabla1(E)* = *5* significa que el apuntador i debe recorrerse al menos 5 caracteres a la derecha antes de que un nuevo intento de coincidencia tenga cualquier esperanza de éxito. Esto es equivalente a recorrer el patrón cuatro caracteres a la derecha, en donde las T coinciden de nuevo.

Sin embargo, *tabla 2* se podría ver como sigue:

1	11
2	10
3	9
4	5
5	4
5	1

De nuevo, el apuntador i debe recorrerse al menos 5 caracteres a la derecha antes de que sea razonable recomenzar el proceso de coincidencia.

Como sucede en este ejemplo, *tabla1(E)* = *tabla2(4)*, lo que implica que el apuntador i debe recorrerse 5 caracteres en cualquier caso. Sin embargo,

tratando otros patrones es posible notar más claramente la utilidad que provee la independencia de las dos tablas.

El algoritmo Boyer-Moore ha sido exhaustivamente probado y rigurosamente analizado. El resultado de las pruebas muestra que en promedio, este algoritmo es "sublineal". Esto es, para encontrar un patrón en la i-ésima posición de una cadena requiere de menos de $i + ancho$ pasos cuando el algoritmo está adecuadamente codificado. En términos del comportamiento en el peor caso, el algoritmo aún se ejecuta en el orden de $i + ancho$.

En cuanto a la utilidad de *tabla 1* y *tabla 2*, es interesante que con alfabetos grandes y patrones pequeños, *tabla 1* resulta más útil. Sin embargo, esta situación se hace inversa para pequeños alfabetos y grandes patrones.

Capítulo 6

Bases de Datos Relacionales

Consultas "Hágalo Usted Mismo"

En muchas aplicaciones que involucran el almacenamiento de información en archivos de computadora, normalmente se dan algunas consultas que un usuario del archivo desea hacer. Por ejemplo, el archivo podría ser un directorio telefónico. Para encontrar el número telefónico de alguien, el usuario del sistema meramente escribe el nombre de la persona en el teclado, y momentos más tarde el correspondiente número telefónico aparece en pantalla. La computadora, usando el nombre como clave o llave, ha buscado a través del archivo por el nombre, recobrado el número telefónico almacenado, y lo despliega en pantalla.

Sin embargo, algunas formas de almacenar datos son más complejas que otras. Involucran muchas clases de información con relaciones especiales entre ellas. Considérese, por ejemplo, los datos asociados con una liga de futbol. Existen varias relaciones entre jugadores y equipos, entre jugadores y números telefónicos, entre equipos y entrenadores, etc. (Figura 6.1).

Para casos como la liga de futbol, se utilizan *bases de datos relacionales*, mediante las cuales es posible producir información que no está almacenada en forma inmediatamente accesible. Puede haber, por ejemplo, una lista de jugadores, sus equipos, y hasta de sus números telefónicos, pero no haber una lista de números telefónicos de todos los jugadores de un equipo en particular [1].

[1] Lo cual sería de mucha utilidad para avisarles si uno de sus juegos resulta cancelado

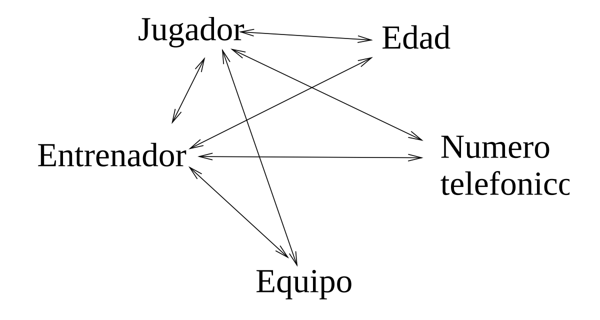


Figura 6.1: Relaciones de datos en una liga de futbol

En una base de datos relacional, es posible crear una "consulta" (*query*) que produzca tal lista (y mucha más información). Para especificar los rangos de consultas posibles en una base de datos relacional, supóngase que la información de la liga de futbol se alamacena en tres tablas separadas:

Tabla de Entrenadores

Equipo	**Entrenador**	**Teléfono**
Leones	M. Cervantes	5535 6798
Demonios	G. González	5671 2033
...	...	...

Tabla de Jugadores

Jugador	**Edad**	**Teléfono**	**Equipo**
A. Johns	22	5432 6592	Leones
L. Echeverría	33	5574 6378	Ángeles
K. Orff	25	5522 1362	Leones
...	...	...	...

Tabla de Encuentros

Local	**Visitante**	**Día**	**Hora**	**Lugar**
Demonios	Ángeles	22 Octubre	12:00 hrs.	Estadio "Julián García"
Leones	Tornados	12 Noviembre	16:00 hrs.	Estadio Nacional
...	...	...	...	...

Cada tabla representa una relación que será usada por la base de datos. Más aún, cada relación es realmente un conjunto de *tuplas*; por ejemplo, la primera relación consiste en tríos de datos con la forma:

Entrenadores:(equipo,entrenador,teléfono)

Las otras relaciones consisten en tuplas de 4 y 5 elementos, respectivamente:

Jugadores:(jugador,edad,teléfono,equipo)
Juegos:(local,visitante,día,hora,lugar)

Cada columna en una tabla (relación) representa un atributo. Algunos atributos funcionan como llaves cuando se busca en las tablas. En las tres tablas anteriores, algunos atributos llave podrían ser *equipo*, *jugador*, *local* y *visitante*.

Hablando matemáticamente, una *relación* es un conjunto de n-tuplas $(x_1, x_2, ..., x_n)$ donde cada elemento x_i se obtiene de un conjunto X_i, y ninguna tupla se repite. Las tablas en una base de datos relacional satisfacen esta definición. Como resultado, la teoría matemática se puede aplicar; en particular, una teoría llamada *álgebra relacional* define operaciones sobre las tablas.

Dos operaciones del álgebra relacional son *selección* y *proyección*. La operación de selección especifica un subconjunto de renglones en una tabla mediante una expresión booleana que involucra atributos. La operación de proyección especifica un conjunto de columnas en una tabla, mediante listar los atributos involucrados. También elimina las duplicaciones que se tengan de los renglones resultantes.

Para obtener una lista de teléfonos de los Leones, estas dos operaciones son suficientes. La selección se denota por σ, y la proyección por π:

$$\pi_{jugadorytelefono}(\sigma_{equipo=Leones}(Jugadores))$$

Analizando de adentro hacia afuera, la operación de selección σ especifica todos los renglones de la tabla de jugadores en el cual el nombre del equipo es Leones. En seguida, de entre estos, la proyección π especifica los atributos

"jugador" y "teléfono", descartando todos los otros atributos de la tupla no especificados por π. Por tanto, la siguiente lista aparece:

A. Johns	5432 6592
K. Orff	5522 1362
...	...

Otra relación importante es llamada la "unión natural" (*natural join*), y se denota con el símbolo $\bowtie$. Opera sobre dos tablas que tienen uno o más atributos comunes. Para cada par de tuplas (una de la primera tabla y una de la segunda), produce una nueva tupla si los atributos comunes tienen valores idénticos para tal par. Supóngase por ejemplo que se le pide a la base de datos relacional la unión de las tablas Entrenadores y Jugadores:

$$Entrenadores \bowtie Jugadores$$

Las dos tablas tienen a *equipo* como un atributo común. Un registro dentro de la nueva tabla (es decir, una relación) que resulta de la operación de unión podría ser:

Leones, M. Cervantes, 5535 6798, A. Johns, 22, 5432 65 92

El atributo *equipo*, que tiene un valor común de Leones para un renglón de Entrenadores y un renglón de Jugadores, resulta en un renglón nuevo en una nueva tabla.

La operación unión funciona de forma combinada con otras operaciones para producir consultas de datos más versátiles. Supóngase que un usuario de la base de datos quisiera una lista telefónica del equipo entrenado por M. Cervantes. La siguiente secuencia de operaciones produciría esa lista:

$$\pi_{jugadorytelefono}(\sigma_{entrenador=M.Cervantes}(Entrenadores \bowtie Jugadores))$$

Dada la tabla Entrenadorer $\bowtie$ Jugadores, la selección de M. Cervantes como entrenador produciría tuplas de 6 elementos conteniendo el atributo *entrenador* con valor de M. Cervantes. Es claro, el valor de Leones sería redundante en este caso. Ya que se requieren solo los nombres y números telefónicos de los jugadores entrenados por M. Cervantes, la proyección se realiza considerando solo estos datos.

Nótese que la misma información es recuperable en muchas formas diferentes de una base de datos relacional. Además, no siempre se obtienen los datos a la misma velocidad. Por ejemplo, la operación unión que se invoca en la consulta anterior es inherentemente ineficiente, ya que se aplica a dos tablas relativamente grandes. Una forma equivalente, pero más eficiente, sería la siguiente:

$$\pi_{jugadorytelefono}(Jugadores \bowtie (\sigma_{entrenador=M.Cervantes}(Entrenadores)))$$

La selección con entrenador = M. Cervantes, cuando se aplica a Entrenadores, produce una sola tupla de tres elementos:

Leones, M. Cervantes, 5535 6798

La unión con Jugadores de esta tupla, entonces, especifica todos los jugadores del equipo de Leones. Una proyección final elimina todos los valores de los atributos excepto *nombre* y *teléfono* de las tuplas resultantes.

Este ejemplo ilustra una característica importante de la clase de lenguajes de consulta de alto nivel que se usan en bases de datos relacionales, y se basa en el álgebra relacional. Tales lenguajes, cuando son completamente implementados, hacen posible a los usuarios optimizar sus consultas. Y dado que hay más de una manera de producir consultas, también es posible seleccionar aquélla que lo hace más rápidamente.

¿Cómo se implementan las bases de datos relacionales? Ya que sólo se mantienen tablas en la memoria de la computadora, todas las operaciones relacionales deben reducirse a operaciones sobre las tablas. El operador selección es fácilmente implementado: simplemente revisa la tabla renglón por renglón, probando una expresión booleana que define los renglones a ser seleccionados a partir de los valores de los atributos de los propios renglones. La proyección también es bastante simple de programar, mediante ordenar los renglones resultantes y eliminar los duplicados.

La operación unión se puede ejecutar más rápidamente en tablas relativamente grandes si las tablas primero se ordenan a partir de los valores del atributo común. Cuando las tablas se ordenan, pueden mezclarse mediante el atributo común, valor por valor.

Hay muchas variaciones para el operador unión. En lugar de igualdad, puede pedirse que los atributos comunes satisfagan otras formas de com-

paración, como por ejemplo, desigualdades. Por otro lado, existen otros operadores además de la selección, proyección y unión, disponibles para las consultas. En la mayoría de los sistemas de bases de datos relacionales, las consultas pueden formarse a partir de operaciones cartesianas de producto, unión, intersección, y substracción de conjuntos.

Hasta la década de los 1970s, se introdujeron tres formas de sistemas de bases de datos: jerárquica, distribuida y relacional. Ahora se reconoce que las bases de datos relacionales son las que tienen la más ámplia variedad de aplicaciones y utilidad. Consecuentemente, sólo se mencionan aquí tal tipo de bases de datos.

Los lenguajes en que las consultas para bases de datos relacionales se expresan se han desarrollado durante varias generaciones de lenguajes, que han ido sofisticándose y refinándose. Las consultas que se han ilustrado hasta ahora tienen una forma algebraica y procedural. Un lenguaje desarrollado para expresar particularmente consultas, llamado SQL, ha sido diseñado para ser usado en forma natural por el humano, y es no-procedural.

La consulta básica en SQL tiene la siguiente forma general:

SELECT $A_1, A_2, ..., A_n$
FROM $R_1, R_2, ...R_m$
WHERE *Expresión Booleana*

Tal consulta implica no sólo la operación selección, sino también la proyección, producto cartesiano, y unión natural. Los argumentos A_i especifican cuáles atributos son parte de la consulta, y los argumentos R_j son relaciones. Los usuarios especifican cuáles relaciones son requeridas para producir una respuesta a una consulta. Ciertamente, la lista $R_1, ..., R_m$ define un producto cartesiano de todas las relaciones (tablas) existentes. La expresión booleana que aparece en la porción "WHERE" de la consulta incluye una variedad de operadores, como por ejemplo, selecciones y uniones.

Ahora es posible mostrar cómo algunas de las consultas anteriores, expresadas usando álgebra relacional, pueden re-expresarse usando SQL. Para producir la lista de los jugadores de los Leones y sus números de teléfono, es posible escribir:

SELECT $jugador, telefono$
FROM $Jugadores$
WHERE $equipo = Leones$

Para el ejemplo donde se involucran una lista de jugadores y sus números telefónicos con el equipo entrenado por M. Cervantes, se podría obtener la consulta de la siguiente forma:

SELECT $jugador, telefono$
FROM $Jugadores, Entrenadores$
WHERE $Entrenadores.entrenador = M.Cervantes$ and
$Entrenadores.equipo = Jugadores.equipo$

Para aquellos programadores que consideran a SQL (o el álgebra relacional) reducido a estructuras relativamente simples, se ofrece el siguiente ejemplo con una consulta anidada:

SELECT $jugador, telefono$
FROM $Jugadores$
WHERE *equipo* In
(SELECT *equipo*
FROM $Entrenadores$
WHERE $entrenador = M.Cervantes$)

Actualmente, SQL se ha convertido en el lenguaje de consultas estándar para los sistemas de bases de datos relacionales de grandes computadoras.

Bibliografía

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[3] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1980.

[4] D.E. Knuth. *The Art of Computer Programming, vol. 1.* Addison-Wesley, 1967.

[5] D.E. Knuth. *The Art of Computer Programming, vol. 3.* Addison-Wesley, 1967.

[6] R.W. Hamming. *Coding and Information Theory.* Prentice-Hall, 1980.

[7] T.A. Standish. *Data Structures and Techniques.* Addison-Wesley, 1980.

[8] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1980.

[9] N. Wirth. *Algoritmos y Estructuras de Datos.* Prentice-Hall, 1987.